GFA
DEBUGGER

BOOK II GFA DEBUGGER

Table of contents

APPENDICES

1. Using the GFA Debugger

1.1. The Editor

The editor of the GFA Debugger is screen oriented, in contrast to many other debuggers. This means that you are no longer required to make all inputs in one 'command' line, but you can move the cursor freely around the screen and correct errors in any line.

The editor also has many valuable functions for easy user input.

Four Debugger screens are available and there is still an additional screen for the program being debugged. You can scroll through listings of the disassembler in dumps, disk directories, and in 'trace back' memory.

1.2 Starting the GFA Debugger

The program disk contains two versions of the GFA Debugger. One is memory resident, the other can be run from the desktop or through the GFA ASSEMBLER.

Memory Resident Debugger

The memory resident debugger may be copied into your Auto folder on your boot disk for automatic loading and installing each time your computer is booted. The debugger captures system errors and gives the user the opportunity to search errors in programs. This memory resident version can be called, by the CALL_DEB.PRG after being installed into memory, or through the GFA ASSEMBLER using the 'Execute Program' function from the Assembler menu.

Executable Debugger

This is the 'normal' version of the debugger and can be started from the desktop or by using the 'Execute Program' function of the Assembler menu from the GFA ASSEMBLER. The GFA Debugger can be passed the name of a program or object file by a command line. Rename the debugger as GFA_DEB.TTP from the GEM desktop, or when calling from the assembler, enter the name of the program in the command line.

When the debugger starts, a check is made to determine if an exception has occurred. The debugger reads the register that would recover the operating system during an exception. This makes it possible to determine what actually caused the exception.

Note: The GFA Debugger requires Trap #6 for internal memory access.

1.3 Keyboard Assignments

The keyboard settings correspond to those of the GFA ASSEMBLER. The following keys are used to move the cursor around the screen:

Cursor up	Moves the cursor up.
Cursor down	Moves the cursor down.
Cursor left	Moves the cursor to the left.
Cursor right	Moves the cursor to the right.

Control + Cursor up
Scroll cursor up.

Control +Cursor down
Scrolls cursor down.
The cursor must be in a line which contains a line from the Memory dump, disassembler listing, disk directory or trace back output for scrolling. Now the cursor can be moved up or down within the corresponding function the same as the text cursor in the editor.

Shift + Cursor up
Moves the cursor to the upper left corner of the display area (Home position).

Shift +Cursor down
Moves the cursor to the lower left corner of the display area.

Shift + Cursor left
Moves the cursor to the beginning of the line.

Shift + Cursor right
Moves the cursor to the end of the line.

Home Moves the cursor to the upper left corner of the display area (Home position).

Shift + Home Clears the screen.

Delete Erase the character under the cursor. All characters to the right of the cursor are moved one character to the left.

Backspace Erase the character to the left of the cursor. All characters to the right of the cursor, including the character under the cursor, are moved one space to the left.

Control + Delete

Erases the line the cursor is currently in.

Control + Cursor right

Erase the line the cursor is in beginning at the present cursor position.

Control + Cursor left

Erase from the start of the line the cursor is presently in up to the present cursor position.

Shift + Control+ Cursor right

Erase the screen starting at the cursor position.

Shift + Control + Cursor left

Erase from the beginning of the screen to the present cursor position.

Insert An empty line is placed in front of the current line.

Shift + Insert Toggle between 'Insert' and 'Overwrite' modes.

Tab The cursor is moved to the next tab position. The tab positions are preset 10 spaces apart. In lines containing a disassembly listing or disk contents the cursor will be moved to the next position with some meaning.

Return Input the current line for execution.
Enter Same as Return.

F1 .. F4 Move between the four debugger screens.

Esc Jump to the screen displaying the program to be debugged. Pressing any key will cause a return to the Debugger Screen. Note: Moving between screens may only be done if nothing has been set with the command 'samescr', because the debugger uses the same screen as 'Execute Program'.

Control + Help The screen contents are printed.

Numeric key pad

You can enter the decimal ASCII code of a character by holding the Shift key and pressing the appropriate keys on the numeric key pad. When the Shift key is released the character will be displayed at the cursor position. All ASCII Characters, between 1 and 255, may be used in your text.

Numeric keys 1 to 9

If you hold the Control key and press a numeric key pad number you can save up to 10 lines, including the cursor position. These lines may be recalled by holding the Control key and pressing the corresponding numeric keypad number. This function enables transferring lines between the four screens of the debugger.

1.3.1 Program Break Keys

These keys enable stopping a running program on various conditions.

Control + Alternate + right Shift

Stop program. With this function you can stop the program that was started from within the debugger or if the debugger is resident in memory the program that is now being executed. Also a program that is 'hung up' and hasn't reset any important system parameters can be stopped using this method.

Left Shift + right Shift

Stop the program at the next observed Trap call (see Section 2.1)

Left Shift + right Shift + Control

Stop the program at the next trap or next breakpoint that is encountered (independent of the contents of the counters). With these last two key combinations it is possible to first create a desired situation in the program and then stop the program at the set position.

Note: The program exit keys are only effective within the program being debugged. This avoids program exits in the operating system and the debugger. As a result, it may be necessary to use the program exit keys more frequently with a program using operating system routines (especially keyboard scans, dialog boxes and similar functions), or alternatively to wait until the processor is unlikely to be in the operating system (following an expected input, for example).

1.4 Mouse Settings

Clicking the left mouse button once will set the keyboard cursor to the mouse cursor position.

Double click left mouse button:

If the mouse cursor is over a label then the memory dump or disassembler listing corresponding to the value of the label will be output beginning with the next line. Labels with an address shown in the text segment are disassembled while other labels are output in a memory dump.

Right mouse button:

A part of a line or the entire line can be marked by holding down the right mouse button. The marked portion of the line is shown in inverse video. When the mouse button is released, the marked line is copied to the current cursor position. In insert mode this area is inserted, in overwrite mode this area will write over the text in the present cursor position. This is how commands for addresses or strings can be expanded.

If the command 'scrollon' (mouse scrolling) is active, the screen will scroll up or downwards when the mouse cursor touches the upper or lower screen edge. This makes it possible to move through memory quickly and comfortably.

2. Debugger Commands

All the commands of the debugger are listed in logical groups. For each command first the syntax, including the possible parameters, is listed followed by a short description of the meaning, an extensive explanation, and when necessary, one or more examples.

Parameters that are shown with brackets {} may be omitted. For parameters that were omitted, default values will be used.

The commands of the debugger can be passed parameter symbols or complex expressions. It is recommended to use a symbol table for testing a program when assembling or linking (see the earlier section of this manual concerning the GFA ASSEMBLER for more information).

Note: The commands 'qon' and 'dir' both use the data buffer. When one of these commands is executed the contents of the buffer is erased. This means that a contents directory read in with 'dir' is usable until another directory is read in or the 'qon' command is executed.

2.1 Debugging

e {'Filename'}{,num}

Loads a program or object code to be debugged into memory (identification of object code is through the filename extension .O), 'filename' is the name of the program that is to be loaded, 'num' is the maximum number of symbols that the user may want to define.

When a program is loaded, the operating system (GEMDOS) reserves the rest of available memory for the program. This memory remains reserved until the program freely gives up some of it. Only one program at a time can be loaded. The debugger itself is not disturbed by the second program.

When object code is loaded it is somewhat different. The object codes are not directly executable because they contain unresolved references (externally defined symbols) and their relocation information is in a different format than that of an executable program. The debugger prepares an 'O' file during loading so that it can be treated like an executable program. Memory in excess of the data length is not required.

Also, undefined symbols can lead to error functions through the isolation of object modules. If the desired program or object module contains a symbol table, the listing will be much more readable during disassembly. If you develop programs you should always use a symbol list during assembly/linking to make debugging easier. These symbols and additional defined symbols can be used in all calculations.

You can learn more about the data format of programs and object modules in Appendix B of the GFA Assembler portion of this manual.

If an ‘e’ is entered in the command line, and the file name is omitted, the GEM Fileselect box will appear for selecting the filename to be loaded. This also applies to the commands ‘is’, ‘fl’, ‘r’ and ‘w’.

Example:

e ,50 Loads a program/O file and sets aside memory for 50 self-defined symbols.

The filename is selected from the GEM fileselector.

ek 'String'

Places a string in the basepage of a loaded program for use as a command line. 'String' is the command line that is to be entered for the program.

Programs that analyze the command line normally use the filename extension .TTP. When a .TTP program is executed from the desktop, a box will appear in which the command line is input. GEMDOS will now automatically load the data name into the command line of the program. A .TTP program may also be an application for a file with a specific extension and be executed by double clicking on one of the specified file types.

These possibilities are not available during debugging. Therefore, the command 'ek' loads a string into the command line of the called program from within the debugger.

The command line of the programs basepage in constructed in the following manner:

length.b	: Length of the command line including the 'cr'
contents.b	: String
13.b	: Carriage return for marking the end of a line.

When using the 'ek' command , only the actual string must be given. The length and the ending carriage return are automatically determined.

Example:

```
e 'editor.prg'      ->Loads the program 'editor.prg'
ek 'test.doc'       ->Loads the command line with test.doc
g                   ->Starts the program
```

ee ‘String’

Loads the address of ‘String’ as the environment in the base page of a loaded program. ‘String’ is a list of strings stored in a buffer and the address is loaded into the basepage of the program as the environment. The strings are separated by a Null byte. The end of the string list is identified by two Null bytes.

Environment strings are used by MS DOS to pass data about the system environment (for example, the data access path) to the program. This is not used by the operating system of the Atari ST. These strings may be used by programmers to pass data between programs.

Example:

One program loads another program and passes this program a parameter, such as the name or address of a data area for the loaded program to use.

v

Displays the parameters of a loaded program or object code.

Important information is available to the editor after loading. The addresses and length of the three segments TEXT, DATA and BSS may be determined from this information. These addresses are often required during debugging. This information can also be determined by using the functions ^^text, ^^data and ^^bss (refer to Section 3).

new

Clear work space.

If a program is read into memory with the 'e' command, all available memory is reserved for this program. No additional programs or object code can be read into memory. The 'new' command releases the work area in order to load a new program.

Because of the many errors in GEMDOS, this command does not always function reliably.

ls {'from'}{,'to'}

List symbols in alphabetical order, 'from' is the letter after which the symbols are to be listed, 'to' is the first character of the symbol representing the stopping point for the listing.

The input of the desired table section is somewhat like that of a telephone book. The output is threefold. Each symbol follows it's value and datatype:

T : Belongs to Text segment
D : Belongs to Data segment
B : Belongs to BSS segment
X : Externally defined
A : Absolute defined
C : Datatype COMMON

See also the corresponding explanations in the Assembler/linker portion of this manual. The output can be paused by pressing the space key and stopped by pressing the <Esc> key.

ln {from,to}

Lists symbols in numerical order, 'from' is the value after which symbols should be listed, 'to' is the value to which the symbols should be listed.

Similar to 'ls', but slightly different in the sort criteria.

diffscr

Two different screens are used for the debugger and the program.

The GFA Debugger is able to differentiate between the screen of the program to be debugged and the screen used by the debugger itself. Therefore the program can be run undisturbed on its own screen. No debugger messages will be displayed on the program screen. The GFA Debugger can even work in medium resolution while it is debugging a program running in low resolution with a different color palette. Switching between screens is as simple as pressing the <Esc> key to go to the program screen and then pressing any key to return to the debugger screen.

samescr

Only one screen is used for the debugger and the program.

In some cases it may be desirable to have the program and the debugger displayed on the same screen. One example would be when a program has little screen output and is to be traced. Switching between screens is no longer necessary.

g {ad}{,breakpoint}

Start program 'ad' is the address at which program execution shall begin. If a starting address is not specified, program execution will begin at the position of the Program Counter, 'breakpoint' is the point at which program execution is to be stopped. If a breakpoint is specified, it is saved as breakpoint 0 (Refer to the commands 'b0–b9='), otherwise no breakpoints will be changed.

This command is used to start a program. If an address is specified, the program will be started at that address, otherwise at the present PC (which can be called through x). By loading a program and displaying the program parameters (by using the command 'v') the PC will be set to the start of the TEXT segment.

The address of a breakpoint can be given as the second parameter. Program execution will stop at this position. A program started with 'g' can be stopped by using a GEMDOS command such as P_TERM (GEMDOS $4C) or through a breakpoint (see command 'b0...9='), a trap (see command 'ob') or through a program break (see Section 1.3.1).

Example:

```
e 'test.prg'   -> Load program
g.position1    -> Start program at the label 'position1' break (Note: the
                  PC is set to the beginning of the TEXT segment with
                  the 'e' command.)
```

c {ad}{,breakpoint}

Call subroutine, then return to the debugger.

The command ‘c’ does almost the same thing as the command ‘g’, but it also places a return address on the stack. This permits a subroutine that ends with ‘RTS’ to be executed.

Example:

```
r ‘test.img’        –> load subroutine to free memory area
c ^^memad           –> call (for ^^... see Section 2.4)
```

bssc

Erase the BSS segment of the program.

This command erases the BSS segment of the loaded program. If the program does not change its TEXT or DATA segments, it can be run from the beginning again after the 'bssc' command. In the case that the program reserves memory using the GEMDOS function 'Mshrink' (GEMDOS $4A) the program should not be executed again. The PC should be set behind the corresponding function call.

Example:

```
e 'test.prg'   -> Load the program
g              -> Start the program
bssc           -> Erase the BSS area
g^^text        -> Start the program from the beginning.
```

initx

Erase the register contents

All data and address registers are erased by this command. The PC, USP, SSP and SR are set to default values.

b0 = ...**b9** = ad{,counter}

Set breakpoint, 'ad' is the address at which program execution is to stop.

'Counter' determines how many times the breakpoint can be passed before program execution will be stopped. If no value is specified, program execution will be stopped at the first encounter of the breakpoint.

A breakpoint is a point within a program at which program execution is to stop. This point may be where the programmer needs to view specific data to find an error. After the breakpoint, the program may be executed one step at a time, in order to localize the error.

Example:

e 'test.prg'	-> Load error filled program
b1=position2,3	-> Program execution shall be stopped after the third encounter of this position.
g,position1	-> Execute program. position1 is now breakpoint 0 and the program execution is stopped at the position 'position1' the first time it is encountered.

In this example the program is stopped at two positions, namely position1, the first breakpoint, or position2, the third breakpoint. By using many unrelated breakpoints an error which could be in one of several routines can be pinpointed.

b

List breakpoints

All ten possible breakpoints are listed. Breakpoints that are not set are shown with an address of $0. If a breakpoint has a counter that does not equal 1 it will be shown behind the corresponding address. The counter is decremented by one each time the corresponding address is encountered. The current counter values are shown in this list (they may be different from the original values), if this program segment has been executed one or more times.

bc

Erase all breakpoints

All set breakpoints are erased. Program execution will no longer be stopped by a breakpoint.

x

Output register contents

The debugger records all the processor registers for the program being debugged. Before a program is started or a single command is executed all registers are passed to the processor. After the program or a command is executed, the registers are again saved. These registers can be viewed using the 'x' command or the 'x list' command. The registers can be changed (see next command).

x list

Input register contents. 'List' is a list of processor registers that are to be changed in order to start a subroutine with specific values, or to correct small errors (false register value).

This command permits any processor registers to be given a new value.

Example:

```
x pc=$12345                                  ; Set PC to $12345
x pc=label1, usp=^^memad+^^memfre, sr=$0300; Set PC to the
                                             value of label1, usp
                                             to the end of
                                             memory, and sr to
                                             $0300.
x d0=12                                      ; Set d0 to dec. 12
x a3=$50000, $12345, ^^resvector             ; Set a3 to $50000,
                                             a4 to $12345, and
                                             a5 to $42a
```

Note: The Registers PC, USP, SSP and SR can only be set in this order. The data register and the address register can be set individually or in increasing order in a command line. Only the first register to be set is given and as many registers are set as there are values. If an attempt is made to set a register higher than d7 or a7 an error message is returned.

t {num}

Trace . 'Num' is the number of commands that are to be traced. If a value is not specified, only one command is executed.

The Trace command is used to test part of a program. The register list is output after each executed program command. If 'num' is not specified, one command (at the current PC position) is executed. If a value is specified for 'num', the corresponding number of commands will be executed.

ta

Auto-trace

Auto trace is an expanded version of the Trace command. The next command in sequence is executed each time the space bar is pressed. If the program is in a subroutine, the subroutine can be executed by pressing the <Return> key. The program will execute the subroutine and the trace command will be executed when control is returned to the calling program. The Auto trace mode can be canceled by pressing the <Esc> key.

regon

Register contents are displayed on the upper screen edge.

This command is a necessary expansion of both the trace commands, displaying a list of the contents of the registers, with their current values, at the upper screen edge. It is much easier to read the value of these registers by executing the program line by line, because the register contents list is shown only once on the screen and the executed commands can be listed in order.

Note: The register contents can be directly changed in this area. If the register is permanently set at the upper part of the screen the function scrolling (see section 1.3) will only be seen in the portion below the register contents. The register contents can be set individually for each of the four debugger screens.

Example:

e 'test.prg'	Load program
regon	Show register contents
ta	Turn on Auto Trace
<space> <space> <space> <space>	Four command executions
<Esc>	End Auto trace

regoff

End display of the register contents.

The area used for viewing the registers is returned.

memon1 Expression1

memon2 Expression2

The memon commands are an expansion of the regon commands: When the Registers are displayed at the upper edge of the screen there will be one or two hexdump memory excerpts at the address represented by 'Expression1' or 'Expression2'. The individual screen areas are divided by horizontal lines.

Memory access can be controlled using this command.

'Expression1' or 'Expression2' allow the same input as the usm command. With '\n' (0<=n<=9) the subconditions defined in us0 ... us9 can be used in the calculation. Inputs so made are compiled into short machine programs as with 'usm' commands to increase execution speed. To check for errors (odd addresses, etc., compare to 'usm') these programs will be executed once when they are input. They will report a system error if an error is encountered.

'memon2' can only be used when 'memon1' is active. The memory block can be turned off through: (a) the regoff command (b) through 'memon1' (turns both areas off) or 'memon2' without an expression. usm and us0...us9 can also be turned off in this way, simply by omitting the expression!

Example:

memon1 l;a1	32 bytes hexdump are presented beginning with the contents of register a1.
ta	Turn on auto trace
<space> <space>	Execute 2 ccmmands beginning at the PC position.

mv

The address specified with memon1 and memon2 can be shown as plain text formulas. (see also 'uv' command).

u {num}

Hide trace (invisible trace), 'num' is the number of commands that are to be invisibly traced. If a value is not specified only one command will be executed.

This command differs from the trace command in that not every command will be followed by a list of the process registers. Only after the specified number of commands have been executed or if execution is stopped are the registers displayed.

qon

Turn trace back on

The debugger stores the commands that have been executed after the hide trace command and can display them at a later time. These commands are saved by using hide trace (see above). Qon erases the trace back buffer, which is rewritten beginning with the next 'u' command.

The GFA Debugger can store up to 4096 commands. It is possible to run a reasonable size program portion and to view the program execution steps later for a better overview of the order in which it (the program) executes them. The commands that have been saved can be viewed by using the 'ql' command (see later).

qoff

Turn trace back off

This command turns off trace back. Commands that have already been saved remain in memory to be viewed later.

ql {from{,to}}

Listing of commands saved in the Trace back buffer. 'From' is the point at which the listing is to begin. Specified as the number of commands. The default value is 0. 'To' is the number of commands which are to be listed. The default value for to is 20.

Any section of the trace back buffer can be listed. After every command in which a PC jump is found (for example through loops, a subroutine call, a return to a program, etc.) an empty line will be output. The scrolling function can be used with the trace back listing (see Section 1.3 and 1.4).

Example:

```
e 'test.prg'  -> Load program
qon           -> Turn on trace back
u 1000        -> Hidden Trace 1000 commands
ql 200        -> List 20 commands beginning with the 200th.
```

usm Expression, **us0** Expression...**us9** Expression

Sets conditions for breaks during hide trace.

By setting break conditions it is possible to direct execution around error filled program areas. If the set break criteria are encountered, the debugger will return the corresponding PC position.

Break conditions are the logical results (true or false) of any complex calculation (with diverse special functions). The input is as with the editor/assembler , in clear text and is automatically compiled and buffered as a short machine language program. The user can list this program and modify it as desired. During program execution in hide trace mode this subprogram will be called after each command is executed. If a value of true(=-1) is returned, a program break will follow. The delay factor involved is dependent on the complexity of the break criteria and increases linearly with the degree of complexity.

SUBCONDITIONS: us0 Expression ... us9 Expression

Expression: Detail of a subcondition. A sub condition is never considered by itself, but only if it is addressed from the main condition. Sub conditions return a Boolean value (-1 or 0, true or false). This is discussed further in the 'usm' command.

NB. USM and USM0..9 can be disabled by not entering an expression after them.

Expression is any formula desired. The following special functions may be placed next to the selectable functions:

1. DIRECT ADDRESSING :

b;address type This is equivalent to the following assembler sequence:

```
move.b  addresstype,d0          ;get value
ext.w   d0                      ;extend to long word
ext.l   d0                      ;result in d0
```

You can find absolute addresses as well as those used in coordination with processor registers. The registers contain the values that are returned from the program being observed. They are not changed by break.

w;address type This is similar to functions like 'b addresstype' with a work width of 'word' and is equivalent to the assembler sequence:

```
move.w  Address_type,d0
ext.l   d0
```

The user must be sure that only even addresses are accessed when addressing memory. Otherwise a bus error, which that will be detected and reported by the debugger, will occur. This also will apply if non accessible addresses are accessed.

l;Address type Comparable to 'b;', 'w;' , assembler sequence:
move.l Address_type,d0

2.INDIRECT ADDRESSING:

b:Address_type Commands that use the 'b;', 'w;', and 'l;' function present a further enlargement of the corresponding 'b;', 'w;', and 'l;' variations. The value itself is of no interest but the contents of the memory lines which contain this value is. In machine language this would be expressed as:

```
move.l   #Value,a0   ; Value=line number.
move.x   (a0),d0     ; Result=contents of memory line
```

=> Value is the argument of an indirect address.

The following applies for the assembler function 'b;':

```
movea.l  Address_type,a0      ; Get the value
move.b   (a0),d0              ; Contents
ext.w    d0                   ; expansion to long
ext.l    d0                   ; result in d0
```

Care must be taken to ensure that address_type does not contain any odd or illegal memory addresses because value will always be treated as a long word. Otherwise a bus error will occur.

w: Address_type Similar to the function 'b:address_type' with the expansion to word width. The equivalent assembler sequence is :

```
movea.l  Address_type,a0      ; Get the value
move.w   (a0),d0              ; contents
ext.l    d0                   ; result in 'd0'
```

l:Address_type This is comparable to 'b:', 'w:'. The equivalent assembler sequence is:

```
movea.l  Address_type,a0      ; get value
move.l   (a0),d0              ; result in d0
```

PRINCIPAL CONDITIONS: usm Expression

Conditional comparison similar to us0 ... us9. There is yet another extended special function to combine the results of conditions:

3. CALLING SUBCONDITIONS:

\0 ... \9	Call of the subcondition us0 ... us9 (return value −1/0, See above)

An interested user can compare the compiled routine with the given break conditions and estimate the amount of time required and use the inline assembler to optimize the routine. Many of the special function addressing-types used in Sections 1 and 2 are not possible, and are emulated.

Example:

(a)

```
usm (\0&\1)|\2
us0 w; d0=100
us1 l;a0=^^physbase
us2 b:(sp)+=%1101
```

This will lead to a program break if the subconditions us0 and us1 or the subcondition us2 is true. The logical operations work bitwise. This makes no difference because the value returned by us0 ... us9 is a boolean value (−1 or 0).

us0 is true only when the lower word from register d0 contains the value 100.

us1 is true only when a0 is shown on the physical screen.

us2 is true only when the upper longword on the stack points to a byte in memory which contains the binary value %1101.

(b)

```
usm(w;(a3)+&w;(a3))=%101010
```

This leads to break if the AND Combination (processor command) of both the addresses specified by a3 return the word %101010. An assembler program would look like:

```
move.w  (a3)+,d0             ; 1. Word
and.w   (a3),d0              ; 2. Word, AND Combination
cmp.w   #%101010,d0          ;result is compared with
                             ; %101010
beq     .Condition_true      ; condition is true
bne     .Condition_false     ; Condition is false
```

(c)

```
us5 w;d5=$10
```

This accomplishes nothing, because the main condition is true.

uv

Show condition.

The break conditions for the Hidden Trace Mode as defined with the commands us0 ... us9 is shown in clear text (formatted). (This text can be modified as desired)

Example:

If we entered the us0 ... us9 and usm definitions as shown in Example (a) as the given conditions, then 'uv' will return the following:

```
usm  (\0&\1)|\2
us0  w;d0=100
us1  l;a0=^^physbase
us2  b:(sp)+=%1101
us3
...
us9
```

ul

Lists those conditions generated in the machine language program.

The compiled machine language programs from the Hidden Trace break conditions are saved in a 4k buffer in the BSS segment of the debugger. This area can be dissassembled using the 'ul' command. The beginning and end of the control subroutine is also named. A break control routine ends with 'rts' and saves all user registers. If the result is true then the zero flag in the CCR Register of the processor is set.

! Assembler Command

An assembler command is executed. This debugger command places an assembler command in a buffer and executes it. This mode is comparable to a command in direct mode of an interpreter in a high level programing language. The result of a single command or the results of a small program can be seen immediately. This command uses all processor registers that can be called with ‘x’ except the PC. The PC remains unchanged.

Example:

```
$F0000: $ ‘Hello, User !’,0      ->Set string
$F0012: ! move.l #$f0000,-(sp) ->Set string address on the stack
$F0012: ! move #9,-(sp)         ->GEMDOS Conws call
$F0012: ! trap #1               ->Call GEMDOS
$F0012: ! addq.l #6,sp          ->Clean up stack
```

Note: $f0000 is an unused address in the work area. All further prompts are given by the debugger.

o list

Observe Trap calls. 'List' is a list of all trap calls which should be observed. Each entry consists of two elements: The trap number and the number of the function.

Legal trap numbers are:

1	GEMDOS
115	GEM VDI
200	GEM AES
13	BIOS
14	XBIOS

The function numbers correspond to the function numbers as documented by Atari. If a value of –1 is passed as the function number, all calls to this trap will be captured.

The 'o' command is implemented to provide an overview of a program or to quickly locate erroneous operating system calls. If an operating system call is found within a program that is one the list of calls to observe, it will be listed with its function number, name, and parameter.

The parameters used with traps 1, 13 and 14 will be shown, along with the corresponding length (word or longword). Parameters containing ASCII characters will also be displayed.

The address of the array is listed next to the function name and number with GEM VDI and GEM AES calls. The GEMDOS functions 'Pterm' ($0), 'Ptermres' ($31) and 'Pterm' ($4c) and GEM function 0 (term), that end a program are captured. If the operating system function generates a return value, this value will be output after the trap is executed.

The trap calls are observed with each debugger controlled execution of a program line (that is with g, c, t, ta and u).

For conditional breaks of the operation system, refer to Section 1.3.1.

Notes: In order to follow the operating system calls of programs with many calls, use the printer protocol 'pon'. Refer to Section 2.4 for more information.

Example:

e 'test.prg'	Load program
pon	Printer protocol on
o 1,-1,200,10,200,19,13,3,14,0,14,10	Observe all GEMDOS traps, GEM AES, appl_init, appl_exit, BIOS Bconout, XBIOS Initmous and Flopfmt.
g	Start program

ob List

Observe with a break after each output

The command 'ob' corresponds to the 'o' command, but the program is stopped after each parameter output and return value. This permits the parameters to be changed as desired. (For example, after disk access an error message could be inserted to test the program in any possible situation.)

Note: If you decide to stop the program while it is running, press both <Shift> keys simultaneously (the key combination <Shift> left + <Shift> Right) will also work (refer to Section 1.3.1).

co

Continue after observe.

Resume program execution after a break in the program. If the program is stopped by a trap call listed in the ob command parameters, the program execution will be interupted and the debugger will report. In order to continue program execution the command 'co' must be entered.

Trace a Trap during program execution:

Without observe:

```
...

trap #n ──────  Exception start  ────────► Routine in S mode
                                                ...
more    ◄─────  Exception end    ────────  rte
```

With observe:

```
trap #n ──  Exception start ────► Observe returns all parameters
                                  ───────► Routine in S mode
                                               ...
                                  Observe returns ◄───  rte
                                  all parameters
more    ◄──  Exception end  ────  back
```

With observe and break:

```
trap #n ──  Exception start ────► Observe returns all parameters
Debugger ◄─────────────────────── back to

.
. (eg. change parameters)
.

<c o >─────────────────────────────────────► Routine in S mode
                                                 ...
                                  Observe returns ◄───  rte
                                  all parameters
Debugger ◄─────────────────────── back to
(eg. change parameters)
.
. End
```

oboff

Turn off observe on break

Observe mode remains active for the specified functions. This command cancels the 'o' command. This can be useful if program execution has been stopped using both shift keys in order to examine a specific trap more closely. The program may then be restarted without further program interuptions. Observe mode may be entered again without passing a new parameter list.

ooff

Turn off observe

The settings of the 'o' command remain in effect until this command turns them off.

2.2 Memory Management

d	{from{,to}}
da	{from{,to}}
dw	{from{,to}}
dl	{from{,to}}

Output memory contents. 'From' is the starting address for the section of memory to be output. 'To' is the ending address of the memory section to be output.

If the ending address is omitted, 11 lines of the memory will be listed.

The four commands d, da, dw and dl output the contents of memory lines. The only difference is the format of the output:

d	Outputs 16 Bytes as Hex numbers and ASCII strings.
da	Outputs 64 Bytes as ASCII strings.
dw	Outputs 8 Words as Hex numbers and as ASCII strings.
dl	Outputs 4 Longwords as Hex numbers and as ASCII strings.

The memory output process can be paused by pressing the space bar and continued by pressing the space bar a second time. It may be stopped completely by pressing the <Esc> key.

The dump of the memory contents may be scrolled by using the scrolling functions (with the cursor keys or mouse. Refer to Section 1.3 and 1.4 for more information.)

Example:

dl 0 -> Long word output of 11 lines beginning at address 0.

dw ^^text -> Word output of 11 lines beginning with the TEXT segment.

da string,string+100 -> 100 byte ASCII output beginning with the Symbol 'string'

d ^^data,^^bss -> Byte output of the DATA segment

$ Bytes, Strings, Words, Longwords

$a Bytes or 'String'

$w Word

$l Longword

Permit values to be stored in memory locations. Only Hex numbers and strings are permitted as input values.

One or more memory locations can be given new values.

The four commands store different values in memory:

value length = 1–2 characters: Byte is stored
value length = 3–4 characters: word is stored
value length = 5–8 characters: long word is stored

If an attempt is made to place a word or a long word at an odd address, an error message will appear.

$a stores one Byte
$w stores two bytes (a Word)
$l stores four bytes (a Longword)

With $a, $w and $l the size of value (byte, word, or long word) doesn't matter. After a line is stored in memory, a prompt with the address of the last value stored will be displayed.

Example:

$50000: $3c,‘hello’,3c2,fc0000,404 -> Stores beginning at address $50000; $3c.b ‘hello’.b $03c2.w $fc0000.l $0404.w

$5000E: $a ‘Hello, here is the GFA Debugger!’,fc0000,404
Stores the bytes ‘Hello, here is the GFA Debugger!’ and 00 and 04 beginning at $5000E

$50032: $w ‘GFA’,0,f8000,10 -> Stores the word 0047,0046,0041,0000,8000 and 0010 beginning at $50032

$5003E: $l f80000,0,’a’,ffc04 -> Stores the longword 000f8000,00000000,00000061 and 000ffc04 beginning at $5003E

Note: The first prompt address ($50000) is an address within free memory. All additional prompt address are automatically calculated by the debugger.

I {from{,to}}

Disassemble

To debug a program it is almost imperative to change the coded program back into clear text to follow the flow of the program. A disassembler listing should return text as close to the original source text of the program as possible. Of course, this is not entirely possible. It's impossible to know if the program was created with the help of macros or with conditional assembly. For this reason the returned translation will be line wise. If the disassembled program contains a symbol table, all the numerical values will be replaced by the proper symbols. Label definition at specific addresses are shown. These symbols are shown in a specialized mnemonic followed by a double period.

Please Note: The form of the disassembler listing can be influenced to some degree by the commands dlon, dloff, cson, csoff, dec, hex, oct, and bin.

dlon

Turns on hex dump during disassembly

A hex dump can be output next to the mnemonics of the assembler command during disassembler. This option is turned on when the debugger is started.

The hex code is enclosed by brackets '["]' and subdivided depending on the function, into a hex word for the opcode and additional word, long words for the operand. The output of hexcode is especially useful when working with symbols if an explicit value is determined in addition to the name of the given address. The text in brackets will not disturb the value of the command with the Inline assembler.

dloff

Deselect 'dlon'

Suppress the output of the hexdump.

cson / csoff

Turn on/off output of small absolute values as symbols.

If a program or object module with a symbol table is loaded, the debugger will assign each number value the first matching symbol found in the table (when possible). If many absolute definitions are contained in a symbol table, each corresponding number value will be replaced.

Because the symbols are used for only a specific position in the program, they could make other sections of the program unreadable. For this reason it is often more informative to use: addq.l #6,sp rather than addq.l #Symbol_xy,sp

hex, dec, oct, bin

Setting of the number base for numerical values during disassembly.

Explanation:

hex : Output in hexidecimal system (base 16)
dec : Output in Decimal system (base 10)
oct : Output in Octal system (base 8)
bin : Output in Binary system (base 2)

[hexcode] {symbol:} Mnemonic {Operand{,Operand}}

or just {symbol:} Mnemonic {Operand{,Operand}}

Inline assembly of individual commands.

Permits direct memory assembly at the prompt address. Input of assembler commands is the same as with the editor/assembler. The commands set for program direction and macro processing (Pseudo Opcodes) are not implemented. Instead all documented system variables and many other important addresses are accessible using function names:

Sytem function syntax:

```
^^ Variable
```

'Variable' can take the following name:

(The value following the equal sign, the return value).

User symbol definitions: (only if the symbol memory is made available with the 'e' command)

At Prompt address : Symbol: {68k ASM command} (The colon can be omitted, if the desired symbol can not be mistaken for a debugger command.)

```
direct      : Symbol =    Expression
        or  : Symbol .equ Expression
              Symbol equ  Expression
              Symbol .set Expression
              Symbol set  Expression
              Symbol ==   Expression
              Symbol .=   Expression
```

Syntax Check:

As with the Assembler/Editor, each command entered is checked before being executed after the Assembler command is entered. Incorrect input is not assembled, but is reported with the corresponding error message.

After the command has been correctly entered, it will be output in control format. The Prompt address knows the address of the last assembled command so that small assembler programs could be entered from within the debugger.

PC Relative Offsets:

Offsets to the PC position are calculated as: Target address = *+Offset

This is especially important with Bcc and Dbcc Mnemonics if they are not used symbolically.

fl from,to, {a}, 'filename'

Reassembly of memory sections or program sections.

The 'fl' directive defines an association between the disassembled listing on the screen and the reassembled program in a data file.

The memory area between 'from' and 'to' is interpreted, tokenised, and then listed in the file specified by 'filename'. Numerical operands (jump lines, immediate addressing, etc.) are replaced by symbols (from the loaded symbol table or automatically created) if their values lie in the range specified by 'from' and 'to'.

The user can also specify if the source text is to be added to an already existing file or put into a newly opened data file. If the first option is desired, the corresponding instruction should have the parameter 'a'. If a new data file is to be created any previous files with the same name will be erased.

Example:

Compare to the example for the 'is' command which follows. A text segment is created.

is 'filename'

Symbolic reassembly of a program/object code.

If the user is analyzing a program written by someone else, it is often desirable to translate a machine code program into source text form in order to change it or further apply it. This process is fully automatic. If a program is loaded into memory using the 'e' command it can be converted into tokenized form, that is, a form readable by the GFA ASSEMBLER.

If the assembler code has its own symbol table, these symbols will be used during the reassembly process. Otherwise generic symbols in the form of 'L Address' will be generated.

'Address' returns the hexidecimal value of the symbol. The source code thus created separates itself into Text, Data, and Bss segments. Jump tables are recognized in the data segments, and saved symbolically as DC.l Pseudo Opcodes. All additional data will be set with DC.b commands.

The subdivisions of the BSS Segment come from the symbol definitions in the BSS Segment and have the form:

```
Symbol1: .DS.b n_1
Symbol2: .DS.b n_2
...
Symboln: .DS.b n_n
```

'n_i' is equal to the number of bytes until the next symbol.

The Source text contains a header with the name of the reassembled file and it's header information. Behind all Mnemonics (text segments) the memory address (from where they were taken) will follow as comments as well as the appropriate hex code. After all .DC.b sequences (data segments) a string with the corresponding ASCII Characters will follow as a comment.

Note: The user should take into consideration that most programs (for example, the GFA ASSEMBLER) are copyright to the author.

Example:

The reassembler listing of a demonstration program without it's own (shortened) symbol table:

```
           ; FILE:D:\SORT\DEMO.PRG
           ;        TEXT      DATA        BSS
           ; BASE: 00041570  00044b12    00045d42
           ; LEN :  000035a2  00001230    00009446

           .TEXT
           bsr     L_0417b4         ; 00041570:  6100  0242
           bsr     L_041f64         ; 00041574:  6100  09ee
           bsr     L_041928         ; 00041578:  6100  03ae
           clr.w   -(sp)            ; 0004157c:  4267
           trap    #$1              ; 0004157e:  4e41
           movea.l#L_045fae,a3      ; 00041580:  267c  00045fae
L_041586:  clr.l   d5               ; 00041586:  4285
           clr.l   d4               ; 00041588:  4284
L_04158a:  move.b$0(a3,d4.w),d3     ; 0004158a:  1633  4000
           bsr     L_0415a4         ; 0004158e:  6100  0014
           asl.l   #$4,d5           ; 00041592:  e985
           or.b    d3,d5            ; 00041594:  8a03
           cmpi.b  #$7,d4           ; 00041596:  0c04  0007
           beq     L_0415a2         ; 0004159a:  6700  0006
           addq.b  #$1,d4           ; 0004159e:  5204
           bra.s   L_04158a         ; 000415a0:  60e8
L_0415a2:  rts                      ; 000415a2:  4e75
...
           .DATA
...
```

```
L_044c20:     .DC.l L_044c44, L_044c5c                ; 00044c20: "........"
              .DC.l L_044c7a, L_044c92                ; 00044c28: "........"
              .DC.l L_044cc1, L_044cee                ; 00044c30: "........"
              .DC.l L_044d17, L_044d42                ; 00044c38: "........"
              .DC.l L_044d6c                          ;  00044c40:  "...."
L_044c44:     .DC.b$2e,$22,$20,$55,$6e,$62,$65,$6b    ; 00044c44: "." Unbek"
              .DC.b$61,$6e,$6e,$74,$65,$72,$20,$42    ; 00044c4c: "annter B"
              .DC.b$65,$66,$65,$68,$6c,$20,$21,$0     ; 00044c54: "efehl !0"
L_044c5c:     .DC.b$2e,$22,$20,$4e,$69,$63,$68,$74    ; 00044c5c: "." Nicht"
              .DC.b$20,$67,$65,$6e,$81,$67,$65,$6e      00044c64" genUgen"
              .DC.b$64.$20,$50,$61.$72,$61,$6d,$65    ; 00044c6c: "d Parame"
....
              .BSS
...
L_045d42: .DS.b $4
L_045d46: .DS.b $4
L_045d4a: .DS.b $2
L_045d4c: .DS.b $2
L_045d4e: .DS.b $2
L_045d50: .DS.b $2
...
              .END
```

hn from,to,ad or **hn** from,to,'String'

Search for an address or a string. 'From' is the starting address for the search, 'to' is the ending address for the search, 'string' is the search string.

The debugger will search the specified area for the expression. All ASCII characters from decimal 1 to 255 may be used in the search string. If the expression is found within the specified area a list will be displayed with the address of every location at which the expression was found.

This list is followed by a memory dump of the first area in which the expression was found. The cursor will be in the middle line of the corresponding byte.

Example:

hn^^data,^^bss,'GFA'	Searches the data segment of the executed program for the string 'GFA'
hn 0,$f8000,^^basepage	Searches the work memory (1MB.) for the address of the basepage of the executed program.

hf from,to,'Command_part'

Searches for a part of an assembler command.

Normally we search for all commands that access a specific memory position or a specific processor register, or we may be interested in a command from which we know only one operand explicitly.

To find such areas use the 'hf' instruction. This function is the opposite of the character string search used in the assembler/editor.

The * wild card is defined as: \

The ? wild card is defined as: ~

The specified search criteria is compared to the disassembler listing from the starting address to the ending address. Characters output through formatting of the disassembler must be taken into consideration. They may be replaced by the \ wild card symbol.

All corresponding lines are listed along with their positions in memory. The search can be paused by pressing the <space> bar and stopped by pressing the <Esc> key.

Example:

debugging a program, input:

```
hf^^text,^^text+$200,"movea.l\,a~"
```

(Search the first 512 Bytes of the text segment for all 'move' commands that write to an address register using longwords.)

The following output will be generated:

```
$BB024:[ 2668 0008          ]           movea.l B_P_L(a0),a3
$BB046:[ 2059               ]           movea.l (a1)+,a0
$BB062:[ 2068 004c          ]           movea.l $4c(a0),a0
$BB0B6:[ 2279 000d3d70      ]           movea.l bildadr,a1
$BB12A:[ 2079 000cbf6c      ]asciiout:  movea.l textpos,a0
$BB158:[ 2279 000d3d74      ]           movea.l bildpos,a1
```

hm from,to,'Command'

Search for a complete assembler command. In contrast to the 'hf' command, the search does not use wild cards or abbreviated assembler commands.

A syntactically correct expression is expected, just as when using the inline assembler. The specified search criteria is transformed into hex code by the inline assembler and searched for in this form.

The result of this is a much faster search than with clear text, which is required when searching large areas of memory.

Example:

```
hm^^text,^^data, trap#14
```

Searches for all trap 14 calls in the text segment and displays all 'hits' in the same format as the 'hf' command.

mc area1,area2

Compare memory areas.

'Area1' and 'area2' are the addresses of the areas that are to be compared. This command will compare any previously loaded data.

The debugger shows a two line memory dump in which the first byte of the difference was found.

Example:

```
r 'data1.xxx',$50000 Read data1 starting at address $50000
r 'data2.xxx',%60000 Read data2 starting at address $60000
mc $50000,$60000 Compare the data and find the first difference.
```

mmc area1,area2

Compare memory areas for different mnemonics. 'Area1' and 'area2' are the addresses of both the areas that are to be compared.

This command is the expanded version of the 'mc' command. The command 'mmc' compares a command word in both areas. That is it compares the first word of 'area1' with the first word of 'area2'.

If a difference is found both the assembler commands found at this address are output. Usually the length of the command is determined and then the command words of the next two commands are compared with each other.

With this command it is possible to compare two programs or object modules in order to determine the difference between an older version and a newer version.

Example:

r 'prog1.prg',$50000	Read program 1
r 'prog2.prg',$60000	Read program 2
mmc $5001c,$6001c	Compare programs after the text segment until the first difference is found.

Note: The 'r' command (See section 2.3) is used because it requires no memory reservation. Therefore as many programs as desired can be loaded to as many addresses as desired.

m from,to,to1

Moves memory area.

'From' is the starting address of the block of memory which is to be moved (the source), 'to' is the ending address of the block of memory which is to be moved. 'To1' is the starting address of the block of memory to which the source is to be moved (the destination). This command moves a memory area intelligently. It can move a memory block a shorter distance than it's own length without hacking off part of the target area. If the area to be moved is greater than the length of the block to be moved, the block will be copied, leaving the source area intact.

Example:

```
m $50000, $60000,$80000 Memory area does not overlap
m $50000, $60000,$55000 Memory area overlaps
m $50000, $60000,$45000 Memory area overlaps
```

mcl from,to

Erase memory area.'from' is the starting address and 'to' is the ending address of the memory block to be erased.

Example:

```
mcl ^^MEMad,^^MEMend      Erase work area
```

fill Number,List

Fill memory area. The fill command initializes a block of memory with a specified numerical pattern beginning at the prompt address.

'Number' specifies how often the numerical pattern defined in 'list' is to be placed in memory.

'List' is a group of expressions expanded with extensions (.b, .w, .l) for the data format (Byte, Word, Long Word) which defines the fill pattern. If the word or long word is set to an odd address the debugger will place it at the next higher even address and leave the odd address undisturbed. The expressions are formatted with all extensions.

Example:

```
$40000: fill 100,"Text to be inserted".b,0.b,^^physbase
```

The string 'Text to be inserted' is stored in memory 100 times starting at the prompt address, $40000. The string ends with a null byte. The next byte remains unchanged in order to reach an even address. Finally the base of the physical screen is inserted as a longword.

2.3 Disk Operations

dir ‘drive:’ or **dir** ‘path’

Display the disk directory.

‘Drive:’ is the disk drive specification for an ‘exact’ directory. ‘Path’ is the file path for output of a directory. If only dir is specified with no path or drive specification then a directory of the default drive will be output.

With this function either an exact contents directory or a normal directory of any path can be output. An exact directory contains, in contrast to a normal directory the start cluster and sector for each data file.

Example:

dir gives an ‘exact’ directory from the current drive

dir ‘c:’ gives an ‘exact’ directory from drive C

dir ‘a:\auto\*.*’ normal directory from the path a:\auto\ .

setdrv 'L' or **setdrv** n

Set the default drive. 'L' is the drive description for the new default drive. 'N' is the number of the new drive (A=0, B=1, C=2, etc.).

The default drive is the drive from which the debugger or the executing program was started. All data input without a specified path will be processed through this drive. This command permits the default drive to be changed from within the debugger. If a program is loaded from one drive and must find data automatically, for example, a resource file or information file, it will only find it if setdrv is executed to change the default drive before the program is started.

Example:

setdrv 'd' or setdrv 3

Both commands set drive d as the new default drive.

delete 'Path'

Erases the file specified by path. 'Path' is the mask of the filename for all files to be erased. The wild card symbols * and ? may be used.

All data with the given specification is erased.

delete 'hello.nix' Erases the file 'hello.nix' in the current directory. delete '\hugo\a*.ttp' Erases all data that begins with the letter 'a' and ends with the extension '.TTP' in the sub directory '\hugo\'.

r 'Data'{,ad}

Reads data. 'Data' is the filename or path that is to be read into memory, 'ad' is the address at which the data is to be placed. If the address is not specified, the data will be stored at the beginning of the free memory area.

This command permits any data to be read into memory. The memory area required is not reserved.

Example:

r 'test.img' The file 'test.img' is loaded starting at the beginning of the free memory area.

w {'data'{,ad,len}}

Write data to disk.

'Data' is the file name or path under which the memory area is to be saved. 'ad' is the beginning address of the memory block to be saved. 'len' is the length of the area to be saved. If data was previously read into memory with 'r' 'Data', 'ad', then the 'len' need not be specified, the data is saved with the parameters used when it was loaded.

This command can be used to write a memory block to the disk.

rv

Display the parameters of the loaded data or sector. If a data file is loaded using 'r' or a set of sectors with 'rsec' (as explained next), this command can be used to display the start address in memory and the length or number of sectors.

rsec {'L',}from,num{,ad}

One or many consecutive sectors are read in. 'L' is the disk drive from which the sectors are read into memory. 'From' is the first sector to read. 'Num' is the number of sectors to read in. 'Ad' is the address where the sectors are to be stored. If a disk drive is not specified the sectors will be read from the default drive. If an address is not specified the sectors will be stored at the start of the free memory.

Example:

rsec 'a',0,1	Reads the boot sector from drive A
d	Displays the first 176 Bytes of the sector

wsec {{'n',}from,num{,ad}}

Write one or many consecutive sectors to disk. See 'rsec' for parameters.

If 'rsec' was previously executed, all parameters can be omitted. The sectors will then be saved with the previous 'rsec' parameters.

Example:

rsec 18,100	Read 100 sectors beginning with sector 18
...	Modify sectors
wsec	Write modified sectors to disk.

or

rsec 'a',0,1440	Read all drive A
;	Change disk
wsec	Copy double sided disk (if enough memory).

secno ad

Calculates sector number and bytes in sector beginning at the specified address, 'ad' is the address from which the sector number is calculated. This command can be used to determine which sector corresponds to what position of the specified address in memory after a number of sectors have been read with rsec.

Example:

rsec 18,10	Read 10 sectors beginning with sector 18
secno $5020e	Determine which sector corresponds to this address.

getbpb {'L'}

Read in Bios BPB. 'L' is the disk drive from which the BPB is to be displayed. If 'L' is not specified, the BPB from the default drive (see setdrv) will be displayed.

This command reads the BIOS Boot Parameter block using BIOS function #7.

Example:

getpbp 'a' Gets the BPB from disk drive A:

sclu {'L'}clu or **sclu** {'L'}–sec

Show cluster status. 'L' is the drive from which the cluster status is to be read. 'Clu' is the cluster from which the status is to be shown. 'Sec' describes a sector and is prefixed with a negative sign. This sector is converted into a cluster from which the status can be shown.

This command makes it possible to check the status of a specific cluster. In order to bypass the conversion of the sector into a cluster, prefix the sector number with a negative value. The following values are possible as a cluster status:

0	The cluster is free (but not necessarily empty).
$ff7	The cluster is defective. (May be $fff7 for a hard disk)
$fff	The cluster is occupied, it is the last cluster of a data file. (May be $ffff with a hard disk)
n	If the cluster is occupied, returns the number of clusters following it.

setclu Nr,value

Set cluster to a new value. Its old value will be displayed and an alert box appears. This instruction can be used, for example, to mark a cluster recognised as damaged, or to 'manually' protect a cluster.

Caution: Use this instruction only with the greatest care, since its mistaken use can cause the loss of valuable data on your disk/hard disk.

If the old cluster values are not identical in the two FAT's, an error message appears and nothing will be changed.

fclu {'L',}beg{,dir}

lclu {'L',}beg{,dir}

Load {lclu} and search {fclu} data file. 'L' is the drive from which the cluster is to be read. 'eg' is the starting cluster from which the data file is to be searched. 'Dir' defines the direction in which the data file is to be searched. If 'dir' is not specified, a positive number is used as the default.

The command 'fclu' searches a data file on the disk. For this a cluster is given at which the search is to begin. If a positive number is used as the mode value, the data is searched beginning with the cluster to its end. If a negative number is used, the data file will be searched from the present location to the start. In this manner you can determine which data file a specific sector belongs to. (Sectors can also be specified by placing a negative prefix in front).

The command 'lclu' loads the cluster found in the forward search.

Example:

fclu 100,-1	Searches the data file to its beginning
lclu 70	70 is the first cluster of the data file. The data file is read into memory.

Note: If clusters are read into memory, the start sector, number of sectors, and the start address in the memory can be displayed with the command 'rv'.

eclu {'L',}beg,num

leclu {'L',}beg,num

Find free clusters (eclu)/ and load them (leclu) into memory. 'L' is the disk drive which is to be searched for the clusters. 'Beg' is first cluster to search for, 'num' is the number of free clusters to searched for.

The command 'eclu' makes it possible to find free clusters on the disk. The beginning cluster and the number of clusters to be found is passed. The debugger sea rches the FAT starting with this cluster and the number of free clusters found is displayed.

The command 'leclu' reads the found clusters into memory.

Example:

A data file that was previously erased needs to be restored. This function may be used if no write or erase functions have been executed on the disk drive in qu estion since the file was deleted. If another data file was erased, the data from that file may be found between the sectors of the data file that is to be restored.

A program on drive C: is erased.

```
setdrv 'c' All commands operations are executed through drive C
  dir        Displays the complete directory of disk in drive C:.
```

The data file is still shown in the directory of the disk, but the first character is now a ‘σ’, meaning that the file has been deleted. Read the starting cluster and length from the displayed directory, and pass this information as the parameters for the ‘leclu’ command:

```
leclu Startcluster,Length/1024+1
```

Note: 1024 is the length of a cluster (for example leclu 1298,9675/1024+1)

Caution: If clusters from other deleted programs are found between the clusters of the program to be restored, those are also read so that all the corresponding clusters are in memory. Then search through the area loaded into memory for clusters that do not belong to the program. This is easy if no other clusters which do not belong to the program to be restored have been loaded.

Foreign sectors can be eliminated by moving all following sectors by using the ‘m’ command (the sector numbers can be calculated using the command ‘secno’ from an address). For example, if sector 60 and 61 do not belong to the program the command:

```
m^(60+2),^^MEMend,^60
```

will move all sectors after sector 62 up two sectors.

After all sectors are in the proper sequence, save the file using the ‘w’ command as a finished program.

2.4 Additional Commands

scrollon, scrolloff

Turns the mouse scrolling function on and off (Refer to Section 1.4).

Note: The default setting for the scrolling function when the debugger is started, is on.

pon, poff

Printer output is enabled (pon) or disabled (poff).

With this command controls the printer. When it is turned on, all output is displayed on the screen as well as being sent to an attached printer.

Comment mark. This command has no effect. It prevents the debugger from trying to evaluate the commands following it.

h Expression

Calculate.

This command performs a mathematical operation and the result is shown in decimal, hexidecimal, binary and octal format.

Example:

```
h^^physbase

h 1+23

h 3 *(7+4)
```

k

The starting and ending address of the area in memory used by the debugger is displayed.

asm {num}

To assembler. 'Num' is the maximum number of new symbols that the user intends to define. This is needed by the assembler if a program is to be written into the debuggers memory.

If the debugger was called from the GFA ASSEMBLER, operation will temporarily return to the assembler. The Assembler will operate normally. The amount of available memory depends on the amount selected when the debugger was called. If a program is assembled or linked in the assembler, it is automatically sent to the memory of the debugger. The assembler remembers that it was called from the Debugger, in displays the text Back to Debugger under the File menu heading. To return to the Debugger, select this function from the menu.

The 'asm' command has two basic purposes:

The source text of the program being debugged may be viewed while working with the debugger. This permits a quicker overview, with the comments and spacing, than provided by the disassembled listing of the debugger.

The assembler can pass programs or object modules directly to the debugger to be tested.

Example:

asm Return to the assembler in order to assemble a program. During assembly a symbol table is to be created. Now exit the assembler by pressing <Esc> + <F1> + < F0> or selecting Back to Debugger from the menu, and the debugger is active again.

v The parameters of the program can be listed in the debugger and the program can be tested.

resident {num}

Exits the Debugger, but the Debugger remains resident in memory. 'Num' is the size of the area that is to be reserved in memory. This command has the same effect as if the resident version of the debugger had been loaded from the auto folder.

quit

Exits the Debugger.

doreset

A system reset is executed.

This command is useful if a program is partly tested and the address of the exit routine of the program is not known or can not be called (the data channel to GEM is closed or other important vectors must be called). A return to the desktop from the Debugger or starting a program may lead to a system crash.

3. Function Analysis

The process of analyzing functions in the debugger is almost identical to that of the assembler. In the debugger the function analysis follows the hierarchical order and permits any number of bracketed levels. This analysis process differs from the Assembler in that:

The debugger does not support macro management. Operations that support macro management are not supported.

The break conditions for the command 'usm' and 'us0' 'us9' can use address types. More exact information can be found in the explanations previously given for these commands.

The following variables are predefined in the debugger:

MEMORY and Color Registers:

```
memconf    =  $ffff8001  ; Memory configuration
dbaseh     =  $ffff8201  ; High byte of the screen address
dbasel     =  $ffff8203  ; Mid byte of the screen address
vcounthi   =  $ffff8205  ; High byte of the video address counter
vcountmid  =  $ffff8207  ; Mid byte of the video address counter
vcountlow  =  $ffff8209  ; Low byte of the video address counter
syncmode   =  $ffff820a  ; Synchronization mode
color0     =  $ffff8240  ; Color registers 0–15
color1     =  $ffff8242
color2     =  $ffff8244
color3     =  $ffff8246
color4     =  $ffff8248
color5     =  $ffff824a
color6     =  $ffff824c
color7     =  $ffff824e
color8     =  $ffff8250
color9     =  $ffff8252
color10    =  $ffff8254
color11    =  $ffff8256
color12    =  $ffff8258
color13    =  $ffff825a
color14    =  $ffff825c
color15    =  $ffff825e
shiftmd    =  $ffff8260  ; Screen Resolution
```

DMA and DISK:

```
Diskctrl   =  $ffff8604  ; Disk Controller Register selection
DMAmode    =  $ffff8606  ; DMA Status/ Mode
DMAhigh    =  $ffff8609  ; DMA Base and counter : high
DMAmid     =  $ffff860b  ; DMA Base and counter : mid
DMAlow     =  $ffff860d  ; DMA Base and counter : low
```

1770 REGISTER:

```
cmdreg      =  $80         ;1770/FIFO Command Register Selection
trackreg    =  $82         ;1770/FIFO Track Register Selection
sectorreg   =  $84         ;1770/FIFO Sector Register Selection
datareg     =  $86         ;1770/FIFO Data Register Selection
```

SOUND CHIP:

```
PSGselect   =  $ffff8800   ; (W) Register selection
PSGread     =  $ffff8800   ; (R) read Data
PSGwrite    =  $ffff8802   ; (W) write data
PSGtoneAf   =  0           ; Channel A: Fine setting
PSGtoneAc   =  1           ; Channel A: Course setting
PSGtoneBf   =  2           ; Channel B
PSGtoneBc   =  3
PSGtoneCf   =  4           ; Channel C
PSGtoneCc   =  5
PSGnoise    =  6           ; Noise generator
PSGmixer    =  7           ; I/O direction, Mixer
PSGampA     =  8           ; Channel A, B, C: Amplitude
PSGampB     =  9
PSGampC     =  $a
PSGenvlpf   =  $b          ; Envelope curve period fine tuning
PSGenvlpc   =  $c          ; Envelope Course tuning
PSGportA    =  $e          ; PORT A (output only)
PSGportB    =  $f          ; PORT B (Centronics output)
```

Bits in "PSGportA":

```
RTSout      =  $8          ; RTS output
DTRout      =  $10         ; DTR output
STROBE      =  $20         ; Centronics strobe output
OUT         =  $40         ; General purpose output
```

68901 (MFP):

```
MFP        = $fffffa00      ; Register Base
MFPgpio    = $fffffa00+1    ; I/O
MFPaer     = $fffffa00+3    ; Active edge
MFPddr     = $fffffa00+5    ; Data direction
MFPiera    = $fffffa00+7    ; Interrupt enable A
MFPierb    = $fffffa00+9    ; Interrupt enable B
MFPipra    = $fffffa00+$b   ; Interrupt pending A
MFPiprb    = $fffffa00+$d   ; Interrupt pending B
MFPisra    = $fffffa00+$f   ; Interrupt in Service A
MFPisrb    = $fffffa00+$11  ; Interrupt in Service B
MFPimra    = $fffffa00+$13  ; Interrupt mask A
MFPimrb    = $fffffa00+$15  ; Interrupt mask B
MFPvr      = $fffffa00+$17  ; Vector Register
MFPtacr    = $fffffa00+$19  ; Timer A control
MFPtbcr    = $fffffa00+$1b  ; Timer B control
MFPtcdcr   = $fffffa00+$1d  ; Timer C & D control
MFPtadr    = $fffffa00+$1f  ; Timer A data
MFPtbdr    = $fffffa00+$21  ; Timer B data
MFPtcdr    = $fffffa00+$23  ; Timer C data
MFPtddr    = $fffffa00+$25  ; Timer D data
MFPscr     = $fffffa00+$27  ; Sync char
MFPucr     = $fffffa00+$29  ; USART control reg
MFPrsr     = $fffffa00+$2b  ; Receiver status
MFPtsr     = $fffffa00+$2d  ; Transmit status
MFPudr     = $fffffa00+$2f  ; USART data
```

ACIA;'s 6850:

```
ACIAkeyctrl = $fffffc00   ; keyboard ACIA control
ACIAkeydr   = $fffffc02   ; keyboard data
ACIAmidicr  = $fffffc04   ; MIDI ACIA control
ACIAmididr  = $fffffc06   ; MIDI data
```

Documented **BIOS VARIABLES** and **LINEA VARIABLES:** (Refer to The Concise Atari ST 68000 Programmer's Reference Guide for more information.) sfc

etv_timer	=	$400
etv_critic	=	$404
etv_term	=	$408
etv_xtra	=	$40c
memvalid	=	$420
memcntlr	=	$424
resvalid	=	$426
resvector	=	$42a
phystop	=	$42e
_membot	=	$432
_memtop	=	$436
memval2	=	$43a
flock	=	$43e
seekrate	=	$440
_timr_ms	=	$442
_fverify	=	$444
_bootdev	=	$446
palmode	=	$448
defshiftmd	=	$44a
sshiftmd	=	$44c
_v_bas_ad	=	$44e
vblsem	=	$452
nvbls	=	$454
_vblqueue	=	$456
colorptr	=	$45a
screenpt	=	$45e
_vbclock	=	$462
_frclock	=	$466
hdv_init	=	$46a
swv_vec	=	$46e
hdv_bpb	=	$472
hdv_rw	=	$476
hdv_boot	=	$47a
hdv_mediach	=	$47e

VPLANES	=	0
VWRAP	=	2
CONTRL	=	4
INTIN	=	8
PTSIN	=	12
INTOUT	=	16
PTSOUT	=	20
COLBIT0	=	24
COLBIT1	=	26
COLBIT2	=	28
COLBIT3	=	30
LSTLIN	=	32
LNMASK	=	34
WMODE	=	36
X1	=	38
Y1	=	40
X2	=	42
Y2	=	44
PATPTR	=	46
PATMSK	=	50
MFILL	=	52
CLIP	=	54
XMINCL	=	56
YMINCL	=	58
XMAXCL	=	60
YMAXCL	=	62
XDDA	=	64
DDAINC	=	66
SCALDIR	=	68
MONO	=	70
SRCX	=	72
SRCY	=	74
DSTX	=	76
DSTY	=	78

_cmdload	=	$482
conterm	=	$484
trp14ret	=	$486
criticret	=	$48a
themd	=	$48e
__md	=	$49e
savptr	=	$4a2
_nflops	=	$4a6
con_state	=	$4a8
save_row	=	$4ac
sav_context	=	$4ae
_bufl	=	$4b2
_hz_200	=	$4ba
_drvbits	=	$4c2
_dskbufp	=	$4c6
_autopath	=	$4ca
_vbl_list	=	$4ce
_dumpflg	=	$4ee
_prtabt	=	$4f0
_sysbase	=	$4f2
_shell_p	=	$4f6
end_os	=	$4fa
exec_os	=	$4fe
scr_dump	=	$502
prv_lsto	=	$506
prv_lst	=	$50a
prv_auxo	=	$50e
prv_aux	=	$512

DELX	=	80
DELY	=	82
FBASE	=	84
FWIDTH	=	88
STYLE	=	90
LITEMSK	=	92
SKEWMSK	=	94
WEIGHT	=	96
ROFF	=	98
LOFF	=	100
SCALE	=	102
CHUP	=	104
TEXTFG	=	106
SCRTCHP	=	108
SCRPT2	=	112
TEXTBG	=	114
COPYTRAN	=	116
SEEDABORT	=	118

DEVICE NUMBERS:

```
PRT =       0 ; Printer
AUX =       1 ; RS 232;
CON =       2 ; Screen ;(vt 52 Emulator)
MIDI =      3 ; MIDI
IKBD =      4 ; Keyboard
RAWCON =  5 ; Screen (raw characters)
```

PROGRAM VARIABLES:

```
basepage      ; Basepage Address of a loaded pogram.
text          ; Text Segment data   - " -
data          ; Data Segment        - " -
bss Segment   ; Bss Segment         - " -
symbols       ; Symbol Table        - " -
MEMfree       ; Beginning of the free memory area (including loaded
              ; program.)
MEMbase       ; Length of the free memory area (including loaded
              ; program.)
MEMend        ; End of the free memory area (including loaded
              ; program.)

physbase      ; physical screen address
logbase       ; logical    -  "  -
```

APPENDIX A

Disk Construction

A disk (or a hard disk partition) is made up of the following parts:

1) Bootsector
2) FAT
3) Contents directory
4) Data

Numbers are saved on the disk in Intel format (backwards, beginning with the low byte) in order to make it easier to exchange the disk with MS DOS format disks.

All information on the disk is divided into sectors. Each sector is typically 512 bytes in length.

1) The boot sector is the first sector on the disk. It contains important data about the disk construction and possibly a boot program with which another program can be executed or the operating system can be loaded from the disk.

Also, if the computer is be booted from a hard disk, the boot partition must contain a small boot program.

The boot sector is constructed as follows:

Offset		Contents	Meaning
$00	0	BRA.S	Jump to Boot program
$02	2	Oems	Reserved for user
$08	8		Serial number 24 Bit Serial number
$0B	11	BPS	Number of Bytes per Sector
$0D	13	SPC	Number of Sectors per Cluster (2)
$0E	14	RES	Number of reserved sectors
$10	16	NFATS	Number of FATs on Disk
$11	17	NDIRS	Number of Directory Entries
$13	19	NSECTS	Number of Sectors on a disk
$15	21	MEDIA	Media Descriptor (not used)
$16	22	SPF	Number of Sectors in a FAT
$18	24	SPT	Number of Sectors per Spur
$1A	26	NSIDES	Number of Sides of Disk
$1C	28	NHID	Number of hidden sectors (not used)
$1E	30	Execflg	Word copied after cmdload
$20	32	Ldmode	if Ldmode=0 load file otherwise read number of Sectors in Sectcnt starting at Ssect
$22	34	Ssect	if Ldmode<>0 load starting here
$24	36	Sectcnt	if Ldmode<>0 Sectcnt read sectors
$26	38	Ldaddr	load address for File
$2A	42	Fatbuf	Address for FAT and DIR Buffer
$2E	46	Fname	Filename: 8 Character Name, 3 character Extension
$39	57		Reserved
$3A	58	Bootcode	Boot program
$01FE	510	CHECKSUM	space used for checksum
$0200	512		

The BIOS Function 7 'Getbpb' and the Debugger command 'getbpb' returns the following data:

BPB.w	Bytes per Sector (512)
SPC.w	Number of Sectors per Cluster (2)
BPC.w	Bytes per Cluster (1024)
SID.w	Number of Sector in Directory
SPF.w	Number of Sector per FAT
SF2.w	Sector number of second FAT
SCL.w	Sector number of the first data clusters
NCL.w	Number of Data Clusters on the disk

2) The data on the disk is divided into clusters on the sectors of the disk. This does not mean that all sectors of the data are in consecutive order. If a file is lengthened, it could happen that data from another file is stored in between the sectors of the newly lengthened file. Therefore there is a list (FAT: File Allocation Table) for every data cluster. The FAT contains the number of the next cluster which continues the data of each file.

This FAT is stored on the disk twice. Both FATs are consecutive on the disk following the boot sector. The operating system utilizes FAT2, but intializes both FATs.

Each FAT entry is 12 bytes long.

Since only 4096 clusters, that is 4 megabytes, can be accessed in 12 bytes, a hard disk must use a 16 byte format. This permits an additional three entries:

0	Cluster is empty
$fff ($ffff on hard disk)	Cluster is last of a data file
$ff7 ($fff7 on hard disk)	Cluster is defective.

The 16 byte entries on the hard drive are very easy to decipher:

Each 16 bit word is in Intel format. Each word in the FAT corresponds to a cluster. The first two cluster numbers (0 and 1) and also the first two FAT entries are not vacated. On the hard disk these entries typically contain the entry 0.

The 12 bit entries are a bit more difficult to read. These entries are always two entries combined with three bytes. Furthermore the entries are bracketed within each other. These entries looks like this:

23 c1 to, the following bits correspond to:

1 :	Bits 8–11	from entry 1
2 :	Bits 4–7	from entry 1
3 :	Bits 0–3	from entry 1
a :	Bits 8–11	from entry 2
b :	Bits 4–7	from entry 2
c :	Bits 0–3	from entry 2

These entries can be read automatically and shown with the command ‘sclu’.

3) The directory contains the name, length, creation time, creation date, start cluster and an attribute byte, which shows that the data is write protected or that it should not appear in the directory for every data file saved. Each entry in the directory is 32 bytes in length and has the following construction:

File name	(8 Bytes)
File Extension	(3 Bytes)
Attribute	(1 Byte)
free Area	(10 Bytes)
Time	(2 Bytes, bitwise)
Date	(2 Bytes, bitwise)
first cluster	(2 Bytes)
Length	(4 Bytes)

This information can be listed with the debugger command 'dir' for each entry.

4) The data, as was explained above, is divided through the clusters of the disk. The start cluster for every file can be found in the directory, all further clusters can be found in the FAT.

Appendix B

Alphabetical list of Debugger commands

Technical Support

Technical support is available to registered users of GFA products. Please ensure that you complete the registration form and return it to GFA. We will only provide support to registered users.

Due to the nature of problems associated with programming, it is preffered that all problems are notified in writing, rather than phone. A source code listing should be provided to support the problem. If possible (and in any case with code longer than 20 lines), a disk containing the source code should be supplied, together with return postage, so the disk and answer can be sent back.

GFA is always improving its products and is always interested in suggested improvements. As a result, upgrades will be available to registered users.

Help and advice is also available from the independent **GFA User Magazine**. The magazine is compiled by GFA users for GFA users. The magazine is available from:

GFA User Magazine
186 Holland Street
Crewe
Cheshire
CW1 3SJ
UK

Contact:
Tel: 0270-256429
Email on CIX (gfa@cix)
Or on the FoReM network of BBS's via MicroMola.

NOTES

GFA Data Media (UK) Ltd
Box 121
Wokingham
Berkshire
RG11 9LP